Section 2: Week 4: Algorithmic System Complexity

Nate Bachmeier

TIM-8110: Programming Languages and Algorithms

May 26, 2019

North Central University

# Algorithmic System Complexity

An algorithm is a well-defined list of steps for performing an action. We use algorithms every day for an assortment of tasks, such as baking a cake. When you bake a cake, it can take anywhere from fifteen minutes to all afternoon. The quality of the produced result can also range from dried up bread to an award-winning masterpiece.

What changes both the time and quality of the result is the implementation of the recipe (algorithm). Just as there are an infinite number of ways to implement a cake, there are infinitely as many implementations for building computer algorithms.

# What is Big-O

The de facto method for measuring the efficiency of a computer algorithm is called Big-O notation. The notation describes the maximum number of steps that are required to perform an action proportional to an input count.

For instance, selection sort has a complexity of O(n^2), which means that to operate across 10 items is at most 100 steps (10\*10). Binary search can be performed in O(log n) complexity and would only take 4 steps to operate across the same items. The improved runtime comes from additional constraints or reducing the number of redundant steps.

Along with the worst-case scenario it is also useful to understand the (1) best case and (2) average case. Consider a simple algorithm to find a specific number within a set. The best case (omega function) occurs when the first item checked matches and there is 1 step required. The average case (theta function) will amortize to n/2 steps.

These three functions can then be used to describe the asymptotically bound range of required steps needed to perform the algorithm (Byrne, 2012) (Cormen & al, 2001). This helps to empirically tell how much each iteration of the algorithm will cost in terms of time.

Looking at the range also helps to drive the conversation around the supported size of N for the component. Even with extremely high complexities such as O(n^n); this can still be acceptable provided N is sufficiently limited. At the same time low complexities such as O(n) can become unusable as N grows into the millions of items.

# Challenges with Big-O

There are many challenges to accurately modeling a system with Big-O notation in practice. The first issue is that the length of a step is not guaranteed to be uniform across a domain, or even within the same problem. Consider the scenario where an array is traversed once, and each element passed to a transform function. The complexity of this this algorithm is equal to the length of the array O(n).

Inside of the transform different permutations are needed to handle different object types, resulting in entirely different code paths. Along these different paths there will be cache misses, resulting in entire bodies of work that will be conditionally performed. Correctly accounting for these nuances requires a white box understanding of the entire implementation. For any reasonably complex production system this is difficult to even white board in its entirety.

Another challenge comes from the assumption the number of steps aligns with time to complete. Assuming each element within the array can be transformed independently then the work could be distributed among N virtual cores. While the number of steps required to complete the work has not been decreased the length of time is now O(N/N=1).

An argument has been made that this is not a realistic scenario as the number of virtual cores is bound. This does not account for cloud scenarios which can provision an unbound number of processors. The major public services also support efficient billing on a per core per second basis. This results in equivalent costs between 100 hours x 1 core or 10 cores for 10 hours. Having the capability to throw unlimited resources at the problem can transition it from a technical challenge to a business decision.

Another issue is that not all algorithmic runtimes are directly influenced by the size of the input set. This can be seen with batch processing systems like Hadoop, where the time to run 100 or 100,000 records is comparable. These high latency and high bandwidth systems take a long time to start, before burning through the input set extremely quickly.

Lastly Big-O notation in practice only focuses on the term with the highest exponent, such as O(4n^2+160n+25) becomes O(n^2) (Sedgewick, 2014). The expression is truncated because n^2 will become the dominate source of the runtime for large values of N.

However, many components only deal with micro batches and N is always a small number. If N is always <=5 then the complexity is mispresented as O(N^2)=25 steps when in actuality it is 925 steps. Additionally, the notation suggests that it is the nested loop that needs to be addressed not the 160n which contains 86% of the runtime.

# Alternatives to Big-O

If the objective of Big-O is to measure time complexity of an algorithm, and the time required for completing an algorithmic system is (1) variable due to interconnected components and (2) influenced by an unbounded resource set; then (3) does this make sense for a modern system to use it? (4) If not, what model would one use?

## QoS Model

An alternative mechanism for measuring complexity is called the Quality of Service (QoS) model, which attempts to measure the efficiency of a system in terms of Transactions Per Second (TPS). This allows for a bottoms-up approach to defining performance characteristics which more closely aligns with business and customer level perceptions (Elliot, 2016).

From the customer’s perspective they care that calling a web page can return the results consistently within one second. They do not have any insight whether the service used 4 million steps to produce that value, nor do they care. They simply want a result within a specified Service Level Agreement (SLA).

The operational team cares about the throughput of each micro service in terms of transactions per second (TPS). TPS is typically counted as the 95th or 99th percentile of concurrent transactions across a scale unit (e.g. production deployment instance). The quality of the TPS is measured in terms of (1) Availability, what percentage of the time was the service responding; (2) Reliability, how many responses were error messages; and (3) Response Time are they finishing within a reasonable duration.

If the TPS are insufficient there are three solutions (a) scale up the size of the resources; (b) scale out the count of resources; or (c) redesign the component. To understand which approach is most efficient the operational team will start by determining what is the limited resource for the component.

Perhaps the memory utilization is very high and scaling up can reduce fragmentation. However, if the component was idle waiting for network responses then adding more compute will not improve the scenario. It simply needs more instances or a more efficient mechanism for waiting on asynchronous I/O.

## Choosing a Solution

Choosing between (a) improve the internals or (b) spending more on cloud resources, becomes a function of the total cost. Perhaps the engineering team reports that it will take two months to redesign the micro service. The engineers on the project would not deliver custom facing defect fixes nor new features during that development time. Then there is an assumption that the improvement will not regress another part of the system. These considerations lead many Internet businesses to accept the tech debt and merely increase cloud resource costs.

## Incorporating Big-O

Some microservices are either part of a critical path or prohibitively expensive to solve through scaling policy. For these scenarios where the internals need to be rewritten a good starting place is with the Big-O complexity of the major code branches. This can help drive the conversation around which areas are prioritized first.

Say that the complexity of a primary code path came back at O(4n^3 + 2n^2 + 16n + 12). This might have been expressed in code as (1) triple nested for-loop; (2) double nested for-loop; (3) loop through results one more time; and (4) in constant time print a result.

The expression (4n^3) will dominate the runtime of this algorithm. It is therefore advantageous to address the triple nested for-loop (1) before focusing on the final loop (3). Perhaps there is a way to leverage a dictionary or similar data structure, and then refactor the nested loops to a lower multiple (NCU, 2016) (Dasgupta, Papadiumitriou, & Vazirani, 2006).

# Conclusions

There are many strengths to measuring system complexity in terms of its ability to economically deliver an SLA. This provides a macro perspective that is easy to align with the business objectives. It can identify which micro services are not meeting their service targets and predict the required scale policy based on measured TPS.

The next layer of detail can be obtained through performance counters, such as memory usage or number of calls to a remote decryption service. This can help us identify which resource is most scarce and needs better management.

Often a resource is inefficiently used because the algorithm has too many steps and is redundantly performing work. Engineering teams can surface this by rewriting the major steps of the code path in terms of Big-O notation. The notation will identify the worst-case scenario and help prioritize which part needs to be improved first.

While this approach enables systems engineers to look at the system from multiple vantage points, it is not intended to be complete replacement to Big-O complexity theory. It is still good practice to understand how the different syntactical structures within the code impact the performance. Even with unlimited cloud resources the components still need to operate on a finite economic budget, and complete within an acceptable duration.

# References

Byrne, B. (2012, August 17). *Big Oh notation (and Omega and Theta).* Retrieved from YouTube: https://youtu.be/ei-A\_wy5Yxw

Cormen, T., & al, e. (2001). *Introduction to Algorithms Second Edition.*

Dasgupta, S., Papadiumitriou, C., & Vazirani, U. (2006). *Algorithms.*

Elliot, K. (2016). Model for Measuring Service Quality.

NCU. (2016). *Algorithm Efficiency, Big-O Notation, and Role of Data.* NorthCentral University.